Report - Praxisblatt 1

Malte A. Weyrich & Jan P. Hummel

May 7, 2024

Disclaimer

Alle Werte die in diesem Report vorgestellt werden, wurden auf einem *Lenovo-ThinkPad-Gen-1* Rechner mit folgender Hardware *AMD® Ryzen 7 pro 4750u with radeon graphics × 16; Memory: 16.0 GB* erhoben. Die Dokumentation zur *usage* der *SMSS JAR* ist auf dem **README.md** des folgenden *GitHub Repository:* `http://https://github.com/github4touchdouble/smss` zu finden.

1 Aufgabe

1.1 Naive Approaches

Der *naive* Ansatz verfolgt eine recht simple Strategie, um alle Segmente in dem Vector C zu identifizieren und die Summe zu bilden. Vom Arbeitsspeicher her ist dieser Ansatz sehr schonend, da die Variablen (wie z.B. *maxscore, l, r* usw.) immer einzeln gespeichert werden und gegebenen Falls überschrieben werden. Jedoch liegt der große Nachteil bei der Laufzeit, welche sich folgender maßen beschreiben lässt:

$$\frac{n^3 + 3n^2 + 2n}{6}$$

Der *rekursive* Ansatz spart sich einige Schritte in der Berechnung, jedoch bleibt der dominante Term weiterhin n^3:

$$\frac{n^3 - n}{6}.$$

Zudem leidet der Stack unter den polynomiell steigenden Funktionsaufrufen, was bereits bei kleinen Eingaben zu Problemen führt. Man könnte also argumentieren, dass obwohl der *rekursive* Ansatz auf dem Papier etwas effizienter ist, für große Eingaben immer noch der *naive* Ansatz am realistischen ist. Beide Algorithmen sind zeitlich eingeschränkt, vor allem ist aber der *rekursive* Ansatz klar limitiert (vom Arbeitsspeicher) was die Eingabegröße n an geht.

In der folgenden Abbildung ist die Laufzeit in Sekunden beider Ansätze gegen die Eingabelänge n grafisch dargestellt:

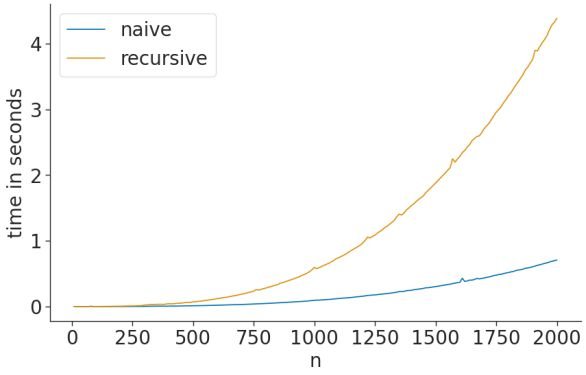

Figure 1: *Naive* algorithm versus *recursive* solution.

Wie sich herausstellt, skaliert der *naive* Ansatz auf der gegebene Hardware besser als der *rekursive* Ansatz. Das liegt wahrscheinlich an den vielen rekursiven Aufrufen, die alle ebenfalls Zeit kosten und daran, wie Java intern mit den Rekursionsaufrufen umgeht.

1.2 Dynamic Programming

Der *dynamic programming* Ansatz bedient sich eines Arrays, welches bereits berechnete Ergebnisse zwischenspeichert und somit redundante Berechnungen verhindert. Somit verbessert sich die Laufzeit stark verglichen zu den oberen Ansätzen:

$$\frac{n^2 + n}{2}.$$

Jedoch bußt dieser Ansatz damit ein, dass der Speicherplatzbedarf quadratisch steigt und das Array jedes mal initialisiert werden muss, was wiederum Zeit benötigt. Zudem ist die Nutzung des Arrays nicht optimal. Es wird jeweils nur die Hälfte des reservierten Speicherplatzes tatsächlich genutzt. In 2.3 wird diese Optimierung umgesetzt.

1.3 Divide and Conquer & Optimal Solution

Der *divide and conquer* hat eine Komplexität von .... TODO JAAAN

Der *optimale* Ansatz löst das Problem in linearer Zeit ($\mathcal{O}(n)$). Genau wie im *naiven* Ansatz werden hier keine zusätzlichen Datenstrukturen verwendet, lediglich werden einzelne Integers gespeichert und überschreiben. Demnach ist der *optimale* Ansatz auch bezogen auf den Arbeitsspeicher sehr effizient.

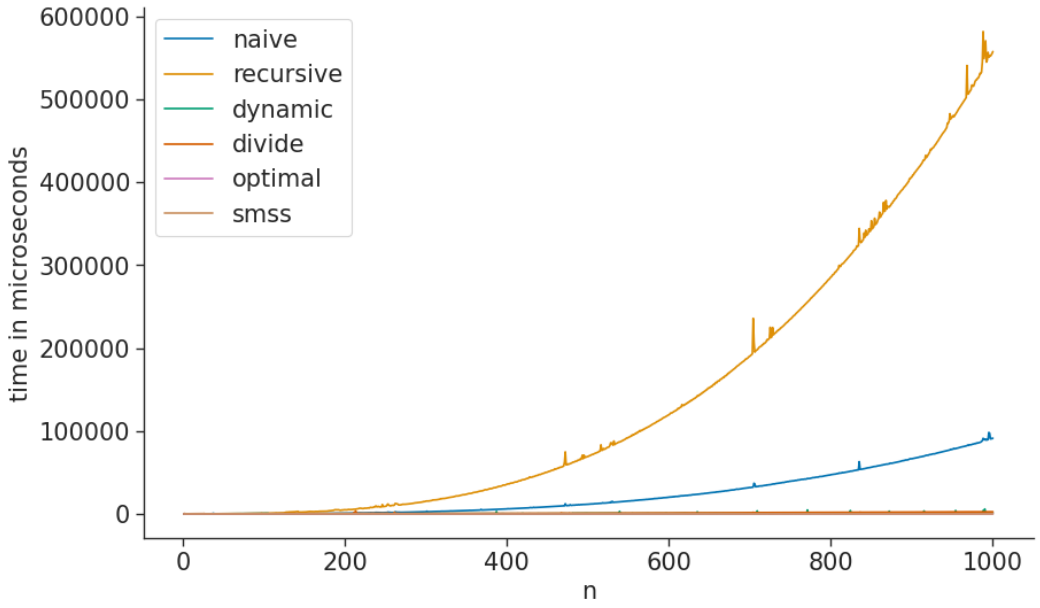

Figure 2: Überblick aller Algorithmen für Eingabeintervall [1; 1000]

2 Aufgabe

2.1 2a - SMSS

Wir erweitern den optimalen Ansatz, indem wir jedes gefundene *segment* in einer *ArrayList< ArrayList<int[3]> >* abspeichern. Die äußere Liste initialisiert für jeden neuen *max_score* der gefunden wurde eine neue innere Liste. Diese innere Liste wird so lange mit gleichwertigen *segmenten* befüllt, bis ein *segment* gefunden wurde, mit einen größerem *max_score*. So ist die Liste von links nach rechts aufsteigend nach *max_score* sortiert (d.h.

die letzte innere Liste beinhaltet alle *MSS*). Am Ende wird einmal über die am *Endindex* stehende Liste in $\mathcal{O}(n)$ iteriert, um die tatsächlichen non overlapping *MSS* in einer finale Liste zu speichern.

Folgende Fälle werden abgedeckt:

```
CASE 1:
1: ---+===+-----------
2: ---+=====+---------
```

Hier darf das $segment_2$ nicht ausgegeben werden, da $segment_1 \subseteq segment_2$. Also merken wir uns jeweils den Start- und Endindex des letzten *segments*. So können wir die letzten Indizes mit den momentanen vergleichen und **CASE 1** abfangen.

```
CASE 2:
1: ---+=====+---------
2: ---+===+-----------
```

CASE 2 ist nicht möglich, da $segment_2$ vor $segment_1$ in der Liste abgespeichert ist. Der Algorithmus ist so konzipiert, dass er wie bei **CASE 1** erst das kürzere $segment_1$ findet und dann das längere $segment_2$. Also müssen wir uns nicht um **CASE 2** kümmern.

```
CASE 3:
1: -----+=+-----------
2: ---+=====+---------

CASE 4
1: ---+=====+---------
2: -----+=+-----------
```

CASE 3 & 4 Sind ebenfalls nicht möglich, da *MSS* entweder den gleichen Start- oder Endindex haben.

```
CASE 5:
1: -----+===+---------
2: ---+=====+---------
```

Hier wird sich in der *i-ten* Iteration die Start- und Endindizes von $segment_1$ gemerkt und demnach das $segment_2$ in der *i+1-ten* Iteration nicht dem Endresultat hinzugefügt, da der Endindex von $segment_2 \not> $ Endindex von $segment_1$.

2.2 2b - SMSS mit minimaler Länge

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

2.3 2c - Alle MSS

Für die 2c haben wir unseren ursprünglichen Ansatz nicht weiter verwendet. Der *optimale Algorithmus* ist in sich selbst so effizient, dass er null Folgen am Anfang und am Ende von Segmenten direkt ignoriert. Wir haben uns also überlegt, welcher der anderen Algorithmen mit eine möglichst guten Laufzeit, diese Fälle auch betrachtet. Wir haben uns dazu entschieden den *dynamic programming Algorithmus* zu erweitern. Dieser hat nämlich den Vorteil bereits ohne Erweiterungen, alle Kombinationen von Teilintervallen zu betrachten. Insbesondere die Teilintervalle, die der *optimale Algorithmus* nicht in Betracht zieht.

Beispiel

```
                        5 -5 9 -3 13 -5 5 -5 5
optimal             :   -----+=====+----------

tatsächliche MSS    :   -----+=====+----------
                        -----+==========+-----
                        -----+================+
                        +==========+----------
                        +==============+-----
                        +====================+
```

Der *dynamic programming* Algorithmus betrachtet in dem oberen Beispiel alle der Teilfolgen. Also haben wir wieder wie in 2.1 eine äußere Liste mit inneren Listen erstellt, und für jeden neuen *maxscore* eine neue innere Liste erstellt, die dann mit allen gleichwertigen Segmenten (also allen Segmenten mit demselben *maxscore*) befüllt wird, bis ein größerer *maxscore* gefunden wird. So findet der *2_c Algorithmus* also definitiv alle MSS, denn wir wissen bereits, dass der *dynamic programming* Ansatz korrekt ist und alle Kombinationen in Betracht zieht. Jedoch bußt dieser Ansatz bei der Speicherkomplexität ein, das *int[][]* Array wächst zu einem gegebenen Eingabe Vektor mit Länge n exponentiell: n^2. D.h. für große Eingaben verbrauchen wir enorm viel Speicherplatz. Wie bereits in 1.2 gezeigt, benutzt der *dynamic programming* Ansatz immer nur die Hälfte des Arrays. Allgemein hat das Array die folgende Form:

$$\begin{pmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & 0 & & \times & \times \\ & & & & \times \end{pmatrix}$$

Beispiel
Betrachten wir die jeweiligen Endzustände der Datenstrukturen von Algorithmus *2_c* und der Arbeitsspeicher schonenden Variante *2_c_1*:

- Eingabe: $v = \{5, -5, 9, -3, 13, -5, 5, -5, 5\}$
- Hardware: ()

```
Array S[n][n] of Default Dynamic Programming (2_c) on v:
   0: [5 ,0 ,9 ,6 ,19,14,19,14,19]
   1: [0 ,-5,4 ,1 ,14,9 ,14,9 ,14]
   2: [0 ,0 ,9 ,6 ,19,14,19,14,19]
   3: [0 ,0 ,0 ,-3,10,5 ,10,5 ,10]
   4: [0 ,0 ,0 ,0 ,13,8 ,13,8 ,13]
   5: [0 ,0 ,0 ,0 ,0 ,-5,0 ,-5, 0]
   6: [0 ,0 ,0 ,0 ,0 ,0 ,5 , 0, 5]
   7: [0 ,0 ,0 ,0 ,0 ,0 ,0 ,-5, 0]
   8: [0 ,0 ,0 ,0 ,0 ,0 ,0 , 0, 5]

Max n on given Hardware: 30808:
java -jar ... --algorithms 2_c --size 30808

ArrayList S containing int[] Arrays of Improved Dynamic Programming (2_c_1) on v:
   0: [5 ,0 ,9 ,6 ,19,14,19,14,19]
   1: [-5,4 ,1 ,14,9 ,14,9 ,14]
   2: [9 ,6 ,19,14,19,14,19]
   3: [-3,10,5 ,10,5 ,10]
   4: [13,8 ,13,8 ,13]
   5: [-5,0 ,-5,0]
   6: [5 ,0 ,5]
   7: [-5,0]
   8: [5]

Max n on given Hardware: 44069:
java -jar ... --algorithms 2_c_1 --size 44069
```

Der *improved dynamic programming 2_c_1* hat keine Position in der Datenstruktur, die ungenutzt bleibt. Der verbesserte Ansatz schafft es 13261 zusätzliche Zahlen zu verarbeiten, also ein $\approx 43\%$ größeres n als der *default 2_c Algorithmus* (*auf der gegebenen Hardware). An der Komplexität hat sich durch die Abänderungen nichts geändert. In Abbildung 3 werden die zwei Ansätze in ihrer Laufzeit verglichen. Die Spikes in der Abbildung entstehen durch die inkonsistente Geschwindigkeit bei der Array Initialisierung.

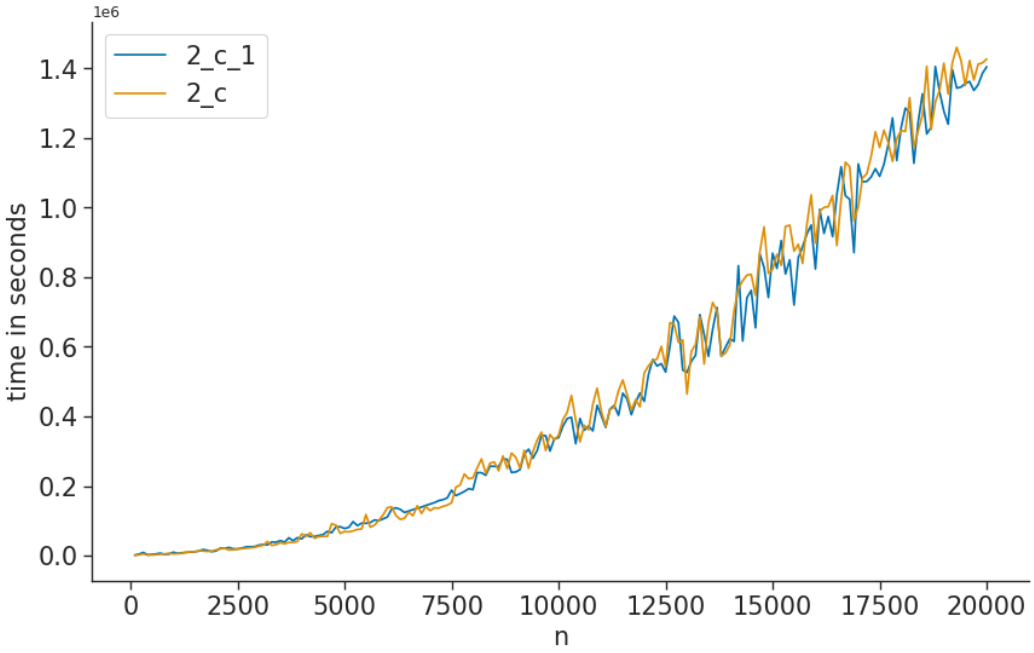

Figure 3: Ansatz 2_c verglichen mit dem Arbeitsspeicher optimierten Ansatz 2_c_1